TITLE

METHOD AND SYSTEM FOR PIPELINED DATABASE TABLE FUNCTIONS

BACKGROUND AND SUMMARY

In a database management system, a function or procedure is a set of statements in a procedural language (e.g., PL/SQL, C, or Java) that is executed as a unit to solve a specific problem or perform a set of related tasks. A function generally differs from a procedure in that the function returns a value to the environment in which it is called. Many database management systems permit users to create and call "table functions" which are a class of functions that produces a set of rows or data objects as output. Table functions can be viewed as a "virtual table" and may be used in the context of a FROM clause (a construct in the database query language SQL) to iterate over individual rows. Further, other SQL or database operations such as selection (e.g., WHERE clause) or aggregation (e.g., GROUP BY clause) can be performed on these rows.

One reason to utilize table functions is to allow users to implement new functions that perform complex transformations on a set of rows or which generates a set of rows, and which are not pre-existing or pre-defined system functions. Another reason for utilizing table functions is to allow users to define functions that operate upon non-native object types in a database system. Non-native object types may be created, for example, using the "CREATE TYPE" command in the Oracle 8*i* database management system from Oracle Corporation of Redwood Shores, California.

Fig. 1 depicts one approach to implementing tables functions in the context of an ETL (Extraction-Transformation-Load) process for data warehousing. The ETL process extracts data

from an online transaction processing (OLTP) system 102, performs a sequence of transformations upon the extracted data, and thereafter loads that transformed data into a data warehouse 108. The transformation steps can be performed by table functions, such as table functions 104 and 106 as shown in Fig. 1. In conventional database systems, a table function cannot accept a pipelined stream of input data; thus, data is usually "staged" before being fed to a table function. In the approach of Fig. 1, the output of a first table function 104 is staged into stage 105 before it is processed by the immediately following table function 106. In effect, the entire output of table function 104 is materialized, possibly into a table or "collection type" instance, before the next transformation table function 106 receives any input data to begin its processing. A collection type describes a data unit made up of a number of elements having a particular data/object type.

Staging forms a blocking operation that presents a significant source of expense and inefficiency to the database system. If the entire output of a first transformation must be materialized before it is passed to the next transformation, excessive memory and/or disk requirements may be imposed because of the staging. Moreover, the overall response time of the ETL operation is affected because each downstream table function remains idle until its corresponding upstream table function has completely constructed the entire set of data that it is to produce. If there is a chain of table functions in the ETL process with staging between each intermediate transformation, then multiple levels of stage-based delays will be imposed. Note that this problem is not limited solely to ETL processes, but exists for many other types of database processes utilizing table functions and staging.

One embodiment of the present invention addresses this problem by implementing input pipelining for table functions. With input pipelining, data does not have to be completely

materialized before it is consumed by a table function. Instead, in one embodiment, a producer of data creates a stream of data that is immediately utilized by a table function consumer of that data. This permits a chain of table functions in which a producer table function generates a pipelined output of data, which is consumed by a consumer table function that accepts a pipelined input of the data. Another aspect of an embodiment is directed to parallel processing of table functions, in which the set of work operated upon by a table function is sub-divided into smaller portions that are assigned to a plurality of database execution slaves. Yet another aspect of an embodiment of the invention is an integration between pipelining and parallelized execution for table functions.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are included to provide a further understanding of the invention and, together with the Detailed Description, serve to explain the principles of the invention.

Fig. 1 depicts an ETL process in which staging occurs between transformations;

Fig. 2 depicts an ETL process in which data is pipelined between transformations;

Fig. 3 shows a flowchart of a process for pipelining according to an embodiment of the invention;

Fig. 4 shows the flow of control for a pipelining process according to an embodiment of the invention;

Fig. 5 depicts chained pipelining according to an embodiment of the invention;

Fig. 6 shows a flowchart of a process for parallel execution of a table function according to an embodiment of the invention;

Figs. 7a and 7b pictorially show range partitioning of work among a plurality of table function slaves according to embodiment(s) of the invention;

Fig. 8 pictorially shows hash partitioning of work according to an embodiment of the invention;

Fig. 9 depicts routing data from a table function to multiple consumers using stages according to an embodiment of the invention;

Fig. 10 depicts pipelining data from a table function to multiple consumers according to an embodiment of the invention;

Figs. 11a and 11b depict pipelining table functions that is executed in parallel according to an embodiment of the invention;

Fig. 12 is a diagram of a system architecture with which the present invention can be implemented; and

Fig. 13 is an additional diagram of a system architecture with which the present invention can be implemented.

DETAILED DESCRIPTION OF EMBODIMENT(S)

PIPELINED TABLE FUNCTIONS

The invention, according to an embodiment, is directed to a system and method for pipelining data into a table function. With input pipelining, data does not have to be materialized completely into a table or collection before it is consumed by a table function. Instead, a producer of data (e.g., a table function or any other source of data) creates a stream of data that is immediately utilized by a table function consumer of that data. Thus, in one embodiment of the invention, a table function can accept one or more streams of rows as input. The rows can be fetched from the one or more input streams on-demand by the table function. In addition, results can be pipelined out of the functions. This enables table functions to be nested or chained together. Further, table functions can be parallelized by partitioning an input stream based on user-defined criteria. This enables table functions to become data transformation entities rather than just data generation units, accepting a streamed data input and transforming it using complex user-defined logic.

The presently disclosed table functions with support for pipelining, streaming, and parallelism represent a new paradigm in solving complex transformation problems inside a database server. The following are some of the benefits offered by such table functions. First, the disclosed table function technology facilitates closer integration of application-specific transformations with the database engine. This is possible because pipelined table functions allow moving compute- and data-intensive user-defined logic from front-end or middle-tier tools into the scalable database server. Second, they help in enabling and speeding up already existing extensible components inside the database, such as spatial joins and text indexing by parallelizing and pipelining table functions. Third, pipelined table functions allow transformation

logic and its related complexity to be pushed into the scalable back-end server such that applications become more ``database-centric", thereby reducing the complexity of front- and middle-tier tools.

Input pipelining for table functions permits a first table function to directly pipeline a stream of data to a second table function. This aspect of the invention is shown in Fig. 2, in which a stream of data subsets 202, 204, and 206 are sent from a first table function 104 ("producer") to a second table function 106 ("consumer"). The second table function 106 can therefore process each data subset, consisting of one or more data objects or rows, as it is streamed from table function 104, thereby avoiding excess wait time and memory consumption expenses as required by the staging approach shown in Fig. 1.

According to one embodiment of the invention, one or more iterators are employed to execute a database statement. An iterator is a mechanism for retrieving rows of data from a database. Associated with each iterator are start, fetch, and close methods. Examples of iterator nodes include table scans, joins, union, etc. Iterator nodes are combined together in a tree that implements a query execution plan based on costing decisions made by an optimizer. At execution time, the execution plan is evaluated in a demand-driven fashion, by fetching the required number of rows from the root node of the iterator tree. In an embodiment of the invention, a table function is internally represented as an iterator in the query execution plan.

Database query language statements, e.g. procedural language statements such as PL/SQL programs, are compiled into machine-independent bytecode in one embodiment of the invention. The bytecode instructions are executed by a PL/SQL Virtual Machine (PVM) at runtime. A PL/SQL engine maintains an execution context for every instantiation of the PVM.

A new instance is created for every invocation of the function. Information like register values, embedded SQL statements, exceptions etc. are stored in the execution context.

In an embodiment, two mechanisms are provided for implementing pipelined table functions. A first approach is referred to as the interface approach. A second approach is referred to as the native database approach. Each of these approaches are described in more detail below.

The interface approach provides flexibility by allowing a table function to be implemented in any procedural language supported by a database system. According to the interface approach, an interface is defined for the creation and pipelined/repeated invocations of table functions and object types having recognized invocations for pipelining. In an embodiment, the interface comprises "Start", "Stop", and "Fetch" interfaces that are embedded into the definition of an object type. The exact contents of these interface routines are specifiable by a user or object type creator to implement desired pipelining functionality. The object type is associated with a table function that is pipeline enabled. When a pipelined table function is executed, the database server accesses these pipelining routines for associated object types at appropriate times to execute the pipelined table function(s). Thus, each object type associated with the pipelining operation corresponds to a set of pipelining routines that define a set of pipelining parameters. The following is example pseudocode for an object type definition under this approach:

```
CREATE TYPE new_object AS OBJECT
(
        object definition statements; /* to define parameters of new object type */

        start ( args )  /* statements to define start routine */
```

```
        start routine statements;

    fetch ( args )  /* statements to define fetch routine */
        fetch routine statements;

    stop ( args ) /* statements to define stop routine */
        stop routine statements;
);
```

In this pseudocode, "new_object" is the name of a new object type, and start(args), fetch(args), and stop(args) refer to start, fetch, and stop command routines respectively.

Fig. 3 depicts a process flow for pipelining according to an embodiment of the interface approach. When a pipelined table function is called, the "start" routine for an associated object type is invoked (302). This start routine comprises one or more statements to perform setup operations required for the pipelining. For example, a context object (or "scan context") can be created and returned to the database server as part of the start routine to maintain state information for the pipelined table function. In an embodiment, the context object is an instance of an object type that models the context to be preserved between successive invocations of the interface routines. Any other desired preliminary operations to setup or prepare the system/data for pipelining can be implemented using the start routine. The following represents an example declaration for a start routine:

```
STATIC FUNCTION TABLESTART ( context_object OUT <scan_context_type>,
<args> )
    RETURN return_type;
```

In this example declaration, "context_object" represents the scan context object set up by the start routine and "args" represents any additional arguments to be passed to the start routine. The

scan context holds the computation state between invocations of the interface routines. It can also be an identifier or a key to a real context that is stored out-of-line.

The database server thereafter invokes the "fetch" routine for the object type to retrieve data produced by the table function (304). The fetch routine for an object type defines the parameters relating to the retrieval and output of data from a pipelined table function that operates upon that object type. For example, the fetch routine defines the quantity of data to retrieve for each fetch invocation. In effect, the fetch routine defines the "degree" of streaming that is performed by a pipelined table function.

The fetch routine returns specified subsets of data consisting of one or more data objects or rows of data. The following is an example of a fetch declaration according to an embodiment of the invention:

MEMBER FUNCTION TABLEFETCH (context_object OUT <scan_context_type>, rows_out OUT <coll-type>) RETURN NUMBER;

In this example declaration, "rows_out" refers to the data fetched by each invocation of the fetch routine and "context_object" refers to the scan context.

The context object created by the start routine is passed as a parameter to the fetch routine. The context object is updated after each fetch invocation to indicate the state of the pipelined table function. If the size of the scan context is large, it may degrade performance due to parameter passing overheads. In such scenarios, it is useful to store the scan context out-of-line, in which the actual scan context resides in the execution environment of the table function and only a handle to the context (e.g., a logical pointer) is passed back and forth.

The fetch routine is repeatedly called until all the data to be outputted by a table function has been returned. According to an embodiment, an end-of-data marker, e.g., a NULL value, is

returned if there remains no more data to be returned by the table function. If the result set returned by the fetch routine is not NULL (306), then the returned data is processed by the database server or consumer (310), and another fetch operation is again iteratively performed to retrieve more data (304). If the result set is NULL, then there is no more data to retrieve and close operations are performed (308).

The number of rows returned for each invocation of the fetch routine can be modified to optimize the table function process. A trade-off between response time and throughput exists with respect to the batch size for results returned from the fetch routine. If a smaller batch size is used, then a faster response time is obtained to fetch an initial set of results. However, smaller batch sizes result a larger number of calls to the fetch routine. Since there may be non-trivial overhead associated with invoking the fetch routine (e.g., if it is implemented as a foreign function), increasing the number of fetch calls may result in greater total execution time. Returning more rows in each invocation of the fetch routine by increasing the batch size reduces the number of fetch calls that must be made, which could improve overall execution times. However, the more rows returned for each fetch, the more delay is imposed before retrieved data is sent to a consumer of the data, causing a slower immediate response time. Larger batch sizes may also lead to increased resource utilization (such as memory resources), thereby degrading scalability. The optimal batch size for a particular application depends upon a variety of factors including the nature of processing within the table function (e.g., CPU intensive or disk intensive), availability of resources (e.g., memory), the cost of function invocation, and the need for good response time versus the need for improved throughput. One approach for optimizing the batch size is to begin with an initial batch size (e.g., 100 rows), and perform experiments on typical workloads to arrive at an appropriately optimal batch size.

Stop operations are executed after completion of the last fetch operation based upon invocation of a "stop" routine for a corresponding object type. The stop routine performs any necessary cleanup operations for the pipelining, such as performing any needed garbage collection for the context object created by the start routine. The following is an example of a stop declaration according to an embodiment of the invention:

MEMBER FUNCTION TABLESTOP (context_object IN <scan_context_type>)
RETURN NUMBER;

Fig. 4 pictorially depicts the flow of control when executing a pipelined table function according to the interface approach in an embodiment of the invention. The process begins when the database server calls a pipeline-enabled table function (402). The table function logic performs any required set-up operations (404) and returns a context object for the function to the database server (406). Fetch operations are iteratively performed as needed to retrieve data generated by the table function (408). Each subset of data is returned to the database server (410). The data may be transferred from the database server to a downstream consumer of data for further processing.

According to the embodiment shown in Fig. 4, processing by the table function logic is halted until the database server makes a next fetch invocation. In this approach, both the producer (e.g., an upstream table function) and consumer of data (e.g., a downstream table function) run on the same execution thread. Producer explicitly returns control back to consumer after producing a set of results. When the consumer is ready for more data, control is returned back to the producer via the fetch invocation. Producer caches the current state (e.g., in a context object) so that it can resume its operations when invoked again by the consumer. According to an alternate embodiment, the table function continues to generate data in parallel to the

processing by consumer, but does not send additional data to the database server until a next fetch call is received. In this approach, both the consumer and producer of data run on separate execution threads and could be on either the same or separate process context. Communications between the consumer and producer of data can be performed, for example, through a pipe or queue mechanism.

Once all the data has been fetched, an end-of-data marker, e.g. a NULL value, is returned to the database server (412). The database server received the NULL marker (413) and thereafter invokes any close operations needed to clean up the pipelining operation (414). The database server thereafter continues its processing (415).

In an embodiment, the consumer of data can pass a callback function to the producer which is executed for each row or data object produced. In this approach, instead of the consumer returning control to the producer, it can call the producer and provide a callback function. The producer then invokes the callback function for each produced row.

According to an embodiment, a table function that implements pipelining is declaratively identified as a pipelined enabled function. The following is an example of a possible declaration that can be employed:

```
CREATE FUNCTION function_id
        { Function body and additional declarations}
PIPELINED USING object_type;
```

Note that any declaration syntax can be utilized within the scope of the invention such that a database system is notified of the existence of a pipelined table function, and the invention is not limited to any specific declaration examples shown herein. This is also true of other

declaration examples shown throughout this document, e.g., for the start, fetch, and stop routines, which can likewise be implemented using other declaration syntax.

The following is an illustrative example of table function declaration pseudocode using the interface approach to convert a row of the type (Ticker, Openprice, Closeprice) into two rows of the form (Ticker, PriceType, Price). So the row ("ORCL", 41, 42) would generates two rows ("ORCL", "O", 41) and ("ORCL", "C", 42).

```
CREATE TYPE BODY StockPivotImpl IS
    -- START routine
    static function Start(sctx OUT StockPivotImpl, p InputCursorType) return
    number is

    begin
        sctx.cur := p; -- save ref cursor in scan context
        return Success; --success code
    end;
    -- FETCH routine
    member function Fetch(outrows OUT TickerTypeSet)return number is
    in_rec StockTable%ROWTYPE;
    idx integer := 1;
    begin
        outrows.extend(100); -- we have chosen to return 100 rows at a time
        while (idx <= 100) loop
            fetch self.cur into in_rec;
            -- first row
            outrows(idx).ticker := in_rec.Ticker;
            outrows(idx).PriceType := "O";
            outrows(idx).price := in_rec.OpenPrice;
            -- second row
            idx := idx + 1;
            outrows(idx).ticker := in_rec.Ticker;
            outrows(idx).PriceType := "C";
            outrows(idx).price := in_rec.ClosePrice;
            idx := idx + 1;
        end loop;
        return Success;
    end;
    -- CLOSE routine (no action required)
    member function Close return number is
```

```
        begin
                return Success;
        end;
end; -- end of type body
```

The disclosed embodiment of the present pipelining process obviates the need for staging data between successive table functions, such as the series of transformations performed during an ETL process. Consistent with one embodiment of the invention, data is consumed by a downstream table function immediately after it is produced by an upstream table function. Since table functions can therefore be symmetrical in both accepting and returning rows of data, table functions can be nested together to form chains of transformations. Fig. 5 illustrates chained pipelining of table functions according to one embodiment of the invention. In Fig. 5, the data output of a first table function 502 is streamed as input into a second table function 504. The data output of the second table function 504 is streamed as input into a third table function 506. An indefinitely long chain of pipelined table functions is thus supportable using the present invention.

A second approach to pipelining is referred to as the native database language approach. In this approach, a native mechanism for pipelining is supported from within a database query language or procedural database query language (such as PL/SQL available from Oracle Corporation or Redwood Shores, California). The native mechanism comprises a recognized command statement that is placed within a pipelined table function. For example, such a command statement could be called a "PIPE" command statement. The following is an example of pseudocode in a procedural database query language for declaring a pipelined table function under this approach:

```
CREATE FUNCTION native_approach (args) RETURN out_data_type
PIPELINED IS data_out out_data_type;
```

```
BEGIN
    FOR in_record IN p LOOP
        processing statements;
        PIPE data_out;
END;
```

The FOR loop in this example can also be represented using the following WHILE loop:

```
WHILE (condition TRUE LOOP processing statements;
PIPE ROW (data-out) processing statements;)
END WHILE
```

In this example pseudocode, the PIPE statement pipelines data (i.e., data_out) out of the table function when it is encountered during execution. In an embodiment, this approach is implemented by allowing both the consumer and producer of data to share the same execution thread. Producer maintains control of the execution thread when it is producing data for output and the producer explicitly gives control to a consumer after producing a set of results. Consumer also explicitly returns control back to the producer when it desires additional data. In operation, a database execution engine, e.g., a SQL engine, takes control when there is a need to retrieve more data from the table function. A database query language interpreter, e.g., a PL/SQL interpreter, takes control after the database execution engine produces a set of data. Control passes from the interpreter to the execution engine until the PIPE command is encountered. The SQL engine consumes the rows and then resumes execution of the PL/SQL interpreter. This procedure continues until all rows are consumed. Similar to the interface approach, a consumer may pass a callback function to the producer to directly execute an operation for each retrieved row of data.

For performance reasons, the database system may batch several rows before providing the output data to a consumer. Alternatively, the consumer can set the number of rows that are output for each PIPE statement. In one embodiment, pipelined table functions have return statements that do not return any values. The return statement can be used to transfer control back to the consumer, to ensure that end-of-data situations are recognized.

The following represents the native database approach to the StockPivot table function set forth above using a procedural database query language (e.g., PL/SQL):

```
create function StockPivot(p InputCurType) return TickerTypeSet
        pipelined is
        in_rec InputRecType;
        out_rec TickerType;
begin
        for in_rec in p loop
                FETCH p into in_rec;
                EXIT when p = "NOTFOUND"; --exit when no more rows
                -- first row
                out_rec.ticker := in_rec.Ticker;
                out_rec.PriceType := "O";
                out_rec.price := in_rec.OpenPrice;
                PIPE row(out_rec);
                -- second row
                out_rec.PriceType := "C";
                out_rec.Price := in_rec.ClosePrice;
                PIPE row(out_rec);
        end loop;
        return;
end;
```

In this example, "InputCurType" corresponds to a ref cursor type. Ref cursors are used by database programming languages to represent a set of rows. These rows can be fetched using the FETCH command.

Under either the interface or native implementation approach, pipelined table functions can be integrated into ordinary database queries (e.g., SQL queries) according to the invention. For example, pipelined table functions can be used in the FROM clause of SQL SELECT statements. As described above, data is iteratively retrieved from a table function as if it is being fetched from an arbitrary query block and can be transparently passed to a consumer of data by the database server. Thus, pipelined table function effectively appears to a database query like any other source of rows or data. Integration of pipelining into standard database query language statements, pursuant to one embodiment of the present invention, allows all the advantages and optimizations that may be applied to any standard query in the database system to be applied to such pipelined statements. For example, standard query optimizations, such as selection of less-expensive execution plan and/or optimized joins, may be applied to optimize the execution of queries comprising pipelined tabled function. Relevant statistics, e.g., cardinality, selectivity, and cost estimation values, may be supplied to an optimizer. These statistics can be invoked by the optimizer to compute the cost of the table functions, allowing the optimizer to generate optimal query plans for queries involving the table functions. Alternatively the optimizer can guess the cardinality (i.e., number of rows) of the table functions and their selectivities with respect to the query predicates.

As another example of integrating pipelined table functions with SQL queries, consider the following SQL statement that pipelines from a table function to SQL (this example references the above StockPivot table function):

```
select * from
table(StockPivot(select * from StockTable));
```

To implement this type of integration according to an embodiment of the invention, a callback function is passed by the query node corresponding to the outer SQL query. In the native language approach, the "PIPE row(out-rec);" statement will then invoke the callback for each row produced. In the interface approach, each "fetch" invocation will execute the callback. For example:

```
select * from
table(StockPivot(select * from StockTable)) x
where x.PriceType = 'C';
```

The callback function passed to 'StockPivot' corresponds to the filter "x.PriceType ='C'". Function 'StockPivot' invokes the filter for each produced row; this helps the producer to eliminate the inappropriate rows without returning them to the consumer. Based on the above-mentioned execution mechanism, rows selected by the query

```
select * from table(StockPivot(...)) where ... ;
```

are displayed as soon as they are produced. In addition, this will benefit statements that pipelined the results from inner sub-query to outer SQL statements:

```
insert into tab select * from table(StockPivot(...)) where ... ;
```

Under certain circumstances, it may be difficult to define the structure of the return type from the table function statically. For example, the structure and form of rows and data objects may be different in different queries and may depend on the actual arguments with which the table function is invoked. Thus, these circumstances make it difficult to know the return type prior to compilation of the table function. According to an embodiment, this issue is addressed by use of dynamic metadata and datatypes to define the return type of a table function.

Under this approach, the return type of a table function is defined using a extensible and definable datatype that can be used to model a collection of data for any returnable or anticipated element or object type. Thus, the table function is declared to return data/collections whose structure is not fixed at function creation time. An exemplary approach to implementing dynamically definable datatypes is disclosed in co-pending U.S. Application Ser. No. _____, attorney docket no. 264/283, entitled "Method and Mechanism for Storing and Managing Self-descriptive Heterogeneous Data," filed on even date herewith, which is hereby incorporated by reference in its entirety.

Since the format of the element may depend on the actual parameters to the table function, the invention provides a Describe interface implemented by the user can be employed in one embodiment of the invention. The database server invokes the Describe routine at query compilation time to retrieve the specific type information for the table function. The Describe routine uses user arguments to determine the shape of the return rows. The format of elements within the returned collection is conveyed to the database server by returning an instance of a datatype to model the metadata of a row. The following is the signature of an example Describe routine:

```
STATIC FUNCTION TableDescribe(rtype OUT ANYTYPE, <args>)
    RETURN NUMBER;
```

To illustrate this aspect of the invention, consider the following table function definition:

```
CREATE FUNCTION AnyDocuments(VARCHAR) RETURN ANYDATASET
PIPELINED USING DocumentMethods;
```

Consider if the following query is placed against the table function:

```
SELECT * FROM
```

```
TABLE(AnyDocuments('http://.../documents.xml')) x
WHERE x.Abstract like '%internet%';
```

Consistent with an embodiment of the invention, the database server invokes the Describe routine at compilation time to determine the return type of the table function. In this example, the DTD of the XML documents at the specified location is consulted and the appropriate AnyType value is returned. The AnyType instance is constructed by invoking the constructor APIs with the field name and datatype information. For the sake of illustration, suppose that the table function could return information on either books or magazines. An example implementation of the Describe routine is as follows:

```
CREATE TYPE Mag_t AS OBJECT
( name VARCHAR(100),
publisher VARCHAR(30),
abstract VARCHAR(1000)
);
STATIC FUNCTION TableDescribe(rtype OUT ANYTYPE,
url VARCHAR)
IS BEGIN
Contact specified web server and retrieve document...
Check XML doc schema to determine if books or mags...
IF books THEN
        rtype=AnyType.AnyTypeGetPersistent('SYS','BOOK_T');
ELSE
        rtype=AnyType.AnyTypeGetPersistent('SYS','MAG_T');
END IF;
END;
```

As mentioned above, during query compilation time, the database server invokes the Describe method and uses the type information (returned via the AnyType out argument) to resolve references in the command line, such as the reference to the "x.Abstract" attribute in the query above. This functionality is particularly applicable when the returned type is a named type, and therefore has named attributes.

Another use of this feature is the ability to describe select list parameters when executing a "select *" query. The information retrieved reflects one select list item for each top level attribute of the type returned by the Describe routine.

Since the Describe method can be called at compile time, the table function in an embodiment has at least one argument which has a value at compile time (e.g. a constant). By using the table function with different arguments, different return types are possible from the function. For example:

```
-- Issue a query for books
SELECT x.Name, x.Author
FROM TABLE(AnyDocuments('Books.xml')) x;

-- Issue a query for magazines
SELECT x.Name, x.Publisher
FROM TABLE(AnyDocuments('Magazines.xml')) x;
```

PARALLELIZING TABLE FUNCTIONS

The present invention, according to one embodiment, also provides a method and mechanism for parallelizing the execution of table functions. When a table function is executed in parallel, the set of data operated upon by the table function is subdivided into smaller portions

to be administered by separate executing entities, such as threads, processes, servers or tasks. These processing entities are referred to herein as "slaves." In many cases, it is highly desirable or even necessary to ensure that the set of work is logically subdivided in a manner that matches the functional characteristics of the tables function, the type(s) of data input/output by the table function, and/or the degree of parallelism configured for the table function. The choice of "partitioning" criteria for subdividing work depends on the logic encoded in the function body, and not all partitioning will produce the correct results for all table functions. In one embodiment, the runtime system should ensure that all data rows/objects with the same partitioned values are handled by the same slave.

As noted above, table functions are often user-defined functions having functionality that is not native to a default database system. Thus, knowledge about the internal operations, behavior, and functionality of table functions are typically more complete for creators of the table function than those that merely invoke or call table functions. In many database systems, a table function appears as a "black box" to the database system, with internal details not completely known or recognized by the database system. Thus, it may be difficult or impossible for the native database parallelization mechanisms to subdivide the work performed by a table function for multiple slaves in a manner that best matches the characteristics or functionality of a table function.

Fig. 6 depicts the process flow of a method 600 to implement parallel execution of a table function according to an embodiment of the invention. A first step of method 600 is to determine the "degree of parallelism" associated with the table function (602). Degree of parallelism refers to the extent to which an amount of work is subdivided into smaller portions.

Many different approaches are appropriate for determining the degree of parallelism associated with a table function.

According to a first approach, the input data source for the table function may be defined to have a specific degree of parallelism that is thereafter adopted by the table function operating upon that data. For example, the base tables that supply data to a table function may be defined to be "parallel" in nature having a specific degree of parallelism. The database system may be configured to recognize metadata associated with base database tables that indicates a preferred degree of parallelism for functions or procedures that operate upon those base tables. In this case, parallelism for a downstream table function is established in a bottom-up manner, by adopting the degree of parallelism associated with its data source. Similarly, if the data source for the table function is a stream of data from another table function that has already been subdivided into a number of parallel execution paths, a convenient degree of parallelism for the downstream table function is one that matches the parallelism of the input data stream.

Alternatively, the database system may establish a degree of parallelism based upon the amount of system resources available for processing the table function. During periods of heavy system activity the database system would establish a lower degree of parallelism, but would establish a greater degree of parallelism during periods of lower system activity. Different levels of priorities could be established for table functions to prioritize the amount of resource assigned to parallelized table functions. Alternatively, the invoker or creator of a table function could statically define the degree of parallelism that is associated with the table function.

An additional step in method 600 is to determine the type of partitioning that should be applied to the input data source (604) to subdivide the work performed by the table function slaves. Without knowing the internal functional details of a table function, it is difficult for a

database system to determine the appropriate partitioning scheme to apply to the input data. Thus, according to an embodiment of the invention, a mechanism is provided to supply information to the database system to indicate a desired partitioning method for the input data. This mechanism permits a specific partitioning method to be chosen for a table function, which is thereafter utilized to subdivided the input data into sets of sub-portions for processing by table function slaves. According to this embodiment, one that is knowledgeable about the internal details of a table function, such as for example the creator of the table function, may specify a desired partitioning method that is embedded into the table function declaration. The following represents an example syntax to specify this information for a table function:

create function f(p refcur_type) return rec_tab_type pipelined
parallel_enable (p **partition by** [<method> (<column list>)]) is
begin ... end;

The "parallel_enable" clause indicates that the function can be executed in parallel. When executing the table function, the "partition by" phrase in the parallel_enable clause specifies the type of partitioning that will be applied to the input data. It also identifies the columns used for partitioning as specified in the <column list> clause.

Any partitioning method may be specified to partition the input data. Examples of suitable partitioning methods include Range, Hash, and "Any" partitioning. Range partitioning decomposes a set of data according to specified ranges of values in a partitioning column. Each range of values is assigned for processing to a separate table function slave. Range partitioning allows rows of data to be partitioned such that the key columns are distinct and monotonically increasing. Hash partitioning decomposes a set of data according to a hash function. One advantage of hash partitioning is that it inherently provides a degree of load balancing for the

work assigned to table function slaves. If the functional behavior of the table function is independent of the partitioning of the input data, the keyword "Any" may be inserted into the Partition By clause to allow the runtime system to choose any appropriate method for partitioning the data. Examples of such functions which are not dependent upon a particular partitioning method include functions which take in one row, manipulate its columns, and generate output row(s) based on the columns of this row only.

Based upon the identified degree of parallelism for the table function and the selected partitioning method, a partitioning definition is generated for the input data to the table function (606). The partitioning definition refers to the partitioning criteria that maps each inbound piece of data to an appropriate table function slave. Each data item in the incoming stream of data is assigned to an appropriate slave (608). The data partitioning process continue until there remains no further work in the incoming data stream (612).

As an example, consider an input stream of data that includes a "Month" column. Assume that the defined degree of parallelism for the table function calls for four separate slaves and that it is desired to partition the input data based upon values in the Month column.

The following represents an example declaration for a table function if Range partitioning is employed:

```
create function f(p refcur_type) return rec_tab_type
        parallel_enable (s partition by RANGE (Month_Column) ) is
begin ... end;
```

According to an embodiment, this function declaration indicates to the database system that range-based partitioning should be utilized to sub-divide the input data, in which the Month column is used as the partitioning column. Since the degree of parallelism calls for four slaves, the input data is partitioned into four separate data portions. In one embodiment, once the type

of partitioning to be applied is selected by the user (e.g. partitioning by range), the database system performs the task of deciding the exact partitioning specifications that should be used to divide the data. In an alternate approach, the user could provide the exact parameters of the partitioning that should be applied. The following represents an illustrative partitioning definition that may be selected, e.g. by a database system, to implement range-based partitioning of input data for the above-example declaration:

```
PARTITION BY RANGE ( Month_Column )
(       PARTITION s1  for values from January and March),
        PARTITION s2  for values from April and June,
        PARTITION s3  for values from July and September,
        PARTITION s4  for values from Octoberand December  );
```

Pursuant to this partitioning criteria, input data corresponding to the first three months of the year ("January", "February", and "March" in the Month column) are assigned to a first partition/slave s1, input data corresponding to the second three months of the year ("April", "May", and "June" in the Month column) are assigned to a second partition/slave s2, input data corresponding to the third three months of the year ("July", "August", and "September" in the Month column) are assigned to partition/slave s3, and input data corresponding to the last three months of the year ("October", "November", "and "December" in the Month column) are assigned to partition/slave s4. Fig. 7a graphically depicts this partitioning definition applied to an inbound data stream. A first processing entity 702 receives the input data stream and applies the partitioning definition to separate the input data into four separate portions. Each data portion is sent to its corresponding slave (s1-s4) for processing.

Since a general partitioning definition has been established, any number of processing entities can be used to separate an incoming stream of data for a set of destination slaves. This is pictorially represented in Fig. 7b, in which two separate processing entities are employed to partition the inbound data and direct the partitioned data portions to corresponding slaves. Each processing entity 708 and 710 need only apply the partitioning criteria to determine which of the table functions slaves s1-s4 correspond to a particular item of input data.

A database system may have a significant scope of discretion with respect to the partition criteria it establishes based upon the defined degree of parallelism and the parameters of the Partition By clause. For example, the above partitioning criteria created partitioning based upon the following ranges: January-March, April-June, July-September, and October-December. For various reasons, e.g., load-balancing, it may be desirable to partition based upon other sets of ranges, such as: December-February, March-May, June-August, and September-November. So long as the database system complies with a desired partitioning method and degree of parallelism, many alternate partitioning criteria could be applied with similar validity to a set of input data.

For this same example, the following represents an example declaration for a table function if Hash partitioning is employed:

```
create function f(p refcur_type) return rec_tab_type
        parallel_enable (p partition by Hash (Month_Column) ) is
begin ... end;
```

In this table function declaration, hash-based partitioning is specified for the input data. Recall that the degree of parallelism defined for the table function calls for a set of four slaves to process the input data. Thus, the following represents example pseudocode for partitioning definition that may be generated for Hash partitioning of the input data:

```
PARTITION BY HASH ( Month_Column )
(       PARTITION into 4 partitions named s1, s2, s3, and s4);
```

Here, four separate partitions have been defined to match the defined degree of parallelism for the table function. Any hash function suitable for a desired hashing result can be employed. The above example partition definition thus only indicates the number of hashing partitions to employ, without necessarily describing the exact hash function to use. Alternatively, a particular hash function may be specified. If a simple round-robin hash function is employed in this example, then every fourth unique data value in the Month column would hash to the portion of data for the same slave. Thus, input data having the values "January", "May", and "September" in the Month column are assigned to partition/slave s1, input data having the values "February", "June", and "October" in the Month column are assigned to partition/slave s2, input data having the values "March", "July", and "November" in the Month column are assigned to partition/slave s3, and input data having the values "April", "August", "and "December" in the Month column are assigned to partition/slave s4.

This partitioning definition is applied to the input data stream to subdivide the input data into appropriate sub-portions. Fig. 8 graphically depicts this partitioning criteria applied to an inbound data stream. A first process entity 802 receives the input data stream and applies the partitioning criteria to separate the input data into four separate portions. Each data portion is sent to its corresponding slave (s1-s4) for processing. Similar to Fig. 7b, any number of processing entities can be employed to separate the incoming data stream into the appropriate sub-portions of data for processing by the appropriate slaves.

The output from multiple slaves may be evaluated in many different ways. For example, the output from multiple slaves could be separated or merged in various combinations before being sent to a next downstream consumer. Multiple downstream slaves acting in parallel may process the parallel outputs from multiple table function slaves.

PARALLEL AND PIPELINED ENABLED TABLE FUNCTIONS

An embodiment of the present invention also allows integration between pipelining and execution of multiple table functions in parallel. Referring to Fig. 9, shown is one approach for directing output data from a first table function to be processed by multiple downstream table functions in parallel. In this approach, the output of a first table function 902 is staged 904 before processing by downstream table functions. Thus, the entire output of table function 902 is materialized into a table or collection, which is then routed as two separate sets of data. Each separate set of data is thereafter sent to two independent invocations 906 and 908 of the same table function for further processing. Table functions 906 and 908 may operate in parallel, since in this example, there are no interactions or data dependencies between these two table functions.

Alternatively, a table function can be pipelined enabled to avoid the staging step 904 as shown in Fig. 9. Referring to Fig. 10, shown is an alternate approach in which a table function 1002 pipelines output data to two separate downstream table functions 1004 and 1006 (which are independent invocations of the same table function). Under this approach, the pipelining mechanism disclosed above is configured such that the retrieved data from a producer table function is sent by the database server to multiple consumers. The database server maintains configuration information pertaining to the number of pipeline consumers that receive data from a particular consumer and routes data to each of these downstream consumers.

In one embodiment, the entire set of output data from table function 1002 is sent to each downstream table function 1004 and 1006. However, a partitioning method may be employed by the database sever to selectively send a subset of the output data to each downstream consumer table function. For example, range based partitioning may be employed to subdivide the output data from table function 1002. Based upon a selected partitioning method, the database server maintains partitioning definition that determines which items of data retrieved from a producer table function should be passed to particular consumers of that data. As each subset of data is fetched from table function 1002, the partitioning definition is applied to determine which of the downstream consumer maps to that data. Each of these data subsets is thereafter routed to the appropriate downstream consumer.

In a preferred embodiment, each consumer runs in a separate execution thread so that the consumers are independent of and in parallel to each other. The producer preferably runs in a separate execution thread from any of the consumers. Alternatively, the producer may share an execution thread with one of the consumer entities.

Fig. 11a shows an embodiment of the invention in which the pipelined output of a first table function 1102 is sent to a table function 1104 that executes in parallel. In this approach, table function 1104 is executed in parallel by slaves 1106 and 1108. As disclosed above a partitioning definition can be created to determine the routing of data sent from table function 1102 to each of the slaves 1106 and 1108. The partitioning definition established for multiple-consumer pipelining is logically similar to the partitioning definition shown above with respect to parallel execution of table functions. The database server utilizes the partitioning definition to determine which of the output data from table function 1102 is sent to the respective slaves 1106 and 1108.

In this approach, each fetch operation performed against table function 1102 retrieves multiple levels of mapping to determine the appropriate consumer. A first mapping determines whether a particular data item is sent to either, or both, of table function 1104 and 1110. If a data item is sent to table function 1104, then the partitioning definition is applied to determine which table function slave 1106 or 1108 corresponds to that data item. Any partitioning method can be employed according to the invention, including hash or rage partitioning as disclosed above. The output of table function 1104 can be merged before being sent to data warehouse 1112. Alternatively, the parallel outputs from slaves 1106 and 1108 can be processed in parallel before insertion into data warehouse 1112.

Fig. 11b shows another embodiment of the invention in which multiple slaves 1122, 1124, and 1126 execute a first table function 1120 in parallel. The output from table function 1120 is sent pipelined and in parallel to a second table function 1140, which also executes in parallel (using slaves 1142, 1144, and 1146) and which outputs a pipelined result. A chain of parallel and pipelined table function can therefore be implemented using this embodiment of the invention. The output of a parallel executed table function 1140 can also be pipelined to a table function 1150 that does not execute in parallel. The output of table function 1150 can thereafter be pipelined to another table function 1160 that executes in parallel.

When a table function fetches rows from one or more input streams, the processing of a stream of rows inside a table function may require that the stream be ordered or clustered on a particular set of keys. This can be required for correctness (e.g. to compute user-defined aggregates, the input rows should be clustered on the grouping keys) or efficiency (e.g. the function has to fetch every row in an input stream only once).

The function writer can specify this behavior of an input stream, according to an embodiment, by extending the function syntax with an ORDER BY or CLUSTER BY clause. Stream ordering works with both serial and parallel table functions. However, for parallel table functions, the ORDER or CLUSTER BY clause should be defined on the same cursor as defined for the PARTITION BY clause, in which only the stream partitioned for parallelism is ordered.

In an embodiment, the ORDER BY clause has the same semantics of global ordering as the ORDER BY clause in SQL. In contrast, the CLUSTER BY clause simply requires that the rows with same key values appear together in the input stream; however, it does not place any restrictions on the ordering of rows with different key values. The ORDER BY or CLUSTER BY is implemented in an embodiment by adding a SORT or HASH iterator between the table function iterator and the top iterator of the query tree corresponding to the input cursor. For parallel table functions, the SORT or HASH iterator is applied on all parallel server processes evaluating the table function after it fetches rows from the input table queue implementing the PARTITION BY.

In an embodiment, the ordering requirement on the input stream is a property of the table function and is specified by the table function implementor. Since the database ensures the appropriate ordering, it places no burden on the invoker of the table function to explicitly order or cluster the input stream to ensure correctness or performance.

New join methods can be implemented by table functions of the present invention that accepts multiple input streams. This is particularly useful in implementing parallel joins of user-defined types. For example, the Oracle Spatial Cartridge, available from Oracle Corporation, defines a Geometry object type and an R-tree indexing scheme. Table functions can be used to implement joins of geometries based on their spatial relationships. Consider the join of two

geometry tables Parks_table and Roads_table using an operator called Overlaps. Assume that R-tree indexes have been created and stored in tables Parks_idx and Roads_idx.

```
SELECT * FROM Parks_table P, Roads_table R
WHERE Overlaps(P.geometry, R.geometry)
```

The WHERE clause can be rewritten to use a table function JoinF() as follows:

```
WHERE (P.rowid, R.rowid) in
(SELECT * FROM TABLE(
  JoinFn(query on root entries of Parks_idx,
      query on root entries of Roads_idx,
      <metadata arguments>)))
```

In this example, JoinF() is a table function that takes a pair of root entries, one each from each R-tree, and joins the subtrees corresponding to them using a nested table function RootJoinFn(). The table function returns the set of satisfying pairs of rowids. The inputs to JoinFn() are queries over the tables that store the R-tree indexes and some associated metadata.

In one embodiment, JoinFn() is implemented as a pipelined and parallel table function using ANY partitioning and a nested-loop join semantics as follows:

```
create function
 JoinFn(p1 RidCurType, p2 RidCurType, metadata)
 return RidPairType pipelined
 parallel_enable(p1 partition by any) is
begin
 for rid1 in p1 loop
  for rid2 in p2 loop
   stmt := "select *
        from TABLE(RootJoinFn(rid1,
```

```
                    rid2, metadata))";
          for each rowid-pair
            returned from execution of "stmt"
            pipe row(<rowid-pair>);
          end for;
        end loop;
      end loop;
      end;
```

Pipelined and parallel-enabled table functions can be used for parallel creation of domain (extensible) indexes. A domain index is an index whose structure is not native to the system and which can be defined by a user. In one approach to implementing user-defined indexes, data that identifies access routines for the user-defined indexes are registered with a database system. In response to relevant statements issued by the database system, the registered routines are called to create or access an index structure relating to the data corresponding to the user-defined index. More information regarding an implementation of domain indexes in described in U.S. Patent 5,893,104, entitled "Extensible Indexing," issued on April 6, 1999, which is hereby incorporated by reference in its entirety.

One approach for creating a domain index includes the following steps: (1) Create table(s) for storing the index data; (2) fetch the relevant data (typically the index key values and row identifiers) from the base table, transform it, and insert relevant transformed data into the table created for storing the index data; and, (3) build secondary indexes on the tables that store the index data, for faster access during query.

If the size of data to be indexed is large, step 2 may provide a performance bottleneck. However, using the present invention, step 2 can be paralleled by modeling it as a parallel table

function (e.g., ParallelIndexLoad()) and invoking this function from an index creation function as follows:

```
INSERT INTO <DomainIndexTable>
SELECT * FROM
 TABLE(ParallelIndexLoad(index_metadata,
      SELECT <key_cols>, <row ids>
      FROM <BaseTable>));
```

SYSTEM ARCHITECTURE OVERVIEW

Referring to Fig. 12, in an embodiment, a computer system 1220 includes a host computer 1222 connected to a plurality of individual user stations 1224. In an embodiment, the user stations 1224 each comprise suitable data terminals, for example, but not limited to, e.g., computers, computer terminals or personal data assistants ("PDAs"), which can store and independently run one or more applications. For purposes of illustration, some of the user stations 1224 are connected to the host computer 1222 via a local area network ("LAN") 1226. Other user stations 1224 are remotely connected to the host computer 1222 via a public telephone switched network ("PSTN") 1228 and/or a wireless network 1230.

In an embodiment, the host computer 1222 operates in conjunction with a data storage system 1231, wherein the data storage system 1231 contains a database 1232 that is readily accessible by the host computer 1222. In alternative embodiments, the database 1232 may be resident on the host computer. In yet alternative embodiments, the database 1232 may be read by the host computer 1222 from any other medium from which a computer can read. In an alternative embodiment, the host computer 1222 can access two or more databases 1232, stored in a variety of mediums, as previously discussed.

Referring to Fig. 13, in an embodiment, each user station 1224 and the host computer 1222, each referred to generally as a processing unit, embodies a general architecture 1305. A processing unit includes a bus 1306 or other communication mechanism for communicating instructions, messages and data, collectively, information, and one or more processors 1307 coupled with the bus 1306 for processing information. A processing unit also includes a main memory 1308, such as a random access memory (RAM) or other dynamic storage device, coupled to the bus 1306 for storing dynamic data and instructions to be executed by the processor(s) 1307. The main memory 1308 also may be used for storing temporary data, i.e., variables, or other intermediate information during execution of instructions by the processor(s) 1307.

A processing unit may further include a read only memory (ROM) 1309 or other static storage device coupled to the bus 1306 for storing static data and instructions for the processor(s) 1307. A storage device 1310, such as a magnetic disk or optical disk, may also be provided and coupled to the bus 1306 for storing data and instructions for the processor(s) 1307. A processing unit may be coupled via the bus 1306 to a display device 1311, such as, but not limited to, a cathode ray tube (CRT), for displaying information to a user. An input device 1312 is coupled to the bus 1306 for communicating information and command selections to the processor(s) 1307. A user input device may include a cursor control 1313 for communicating direction information and command selections to the processor(s) 1307 and for controlling cursor movement on the display 1311.

According to one embodiment of the invention, the individual processing units perform specific operations by their respective processor(s) 1307 executing one or more sequences of one or more instructions contained in the main memory 1308. Such instructions may be read into the

main memory 1308 from another computer-usable medium, such as the ROM 1309 or the storage device 1310. Execution of the sequences of instructions contained in the main memory 1308 causes the processor(s) 1307 to perform the processes described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and/or software.

The term "computer-usable medium," as used herein, refers to any medium that provides information or is usable by the processor(s) 1307. Such a medium may take many forms, including, but not limited to, non-volatile, volatile and transmission media. Non-volatile media, i.e., media that can retain information in the absence of power, includes the ROM 1309. Volatile media, i.e., media that can not retain information in the absence of power, includes the main memory 1308. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise the bus 1306. Transmission media can also take the form of carrier waves; i.e., electromagnetic waves that can be modulated, as in frequency, amplitude or phase, to transmit information signals. Additionally, transmission media can take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications. Common forms of computer-usable media include, for example: a floppy disk, flexible disk, hard disk, magnetic tape, any other magnetic medium, CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, RAM, ROM, PROM (i.e., programmable read only memory), EPROM (i.e., erasable programmable read only memory), including FLASH-EPROM, any other memory chip or cartridge, carrier waves, or any other medium from which a processor 1307 can retrieve information.

Various forms of computer-usable media may be involved in providing one or more sequences of one or more instructions to the processor(s) 1307 for execution. For example, the instructions may initially be provided on a magnetic disk of a remote computer (not shown). The remote computer may load the instructions into its dynamic memory and then transit them over a telephone line, using a modem. A modem local to the processing unit may receive the instructions on a telephone line and use an infrared transmitter to convert the instruction signals transmitted over the telephone line to corresponding infrared signals. An infrared detector (not shown) coupled to the bus 1306 may receive the infrared signals and place the instructions therein on the bus 1306. The bus 1306 may carry the instructions to the main memory 1308, from which the processor(s) 1307 thereafter retrieves and executes the instructions. The instructions received by the main memory 1308 may optionally be stored on the storage device 1310, either before or after their execution by the processor(s) 1307.

Each processing unit may also include a communication interface 1314 coupled to the bus 1306. The communication interface 1314 provides two-way communication between the respective user stations 1224 and the host computer 1222. The communication interface 1314 of a respective processing unit transmits and receives electrical, electromagnetic or optical signals that include data streams representing various types of information, including instructions, messages and data. A communication link 1315 links a respective user station 1224 and a host computer 1222. The communication link 1315 may be a LAN 1226, in which case the communication interface 1314 may be a LAN card. Alternatively, the communication link 1315 may be a PSTN 1228, in which case the communication interface 1314 may be an integrated services digital network (ISDN) card or a modem. Also, as a further alternative, the communication link 1315 may be a wireless network 1230. A processing unit may transmit and

receive messages, data, and instructions, including program, i.e., application, code, through its respective communication link 1315 and communication interface 1314. Received program code may be executed by the respective processor(s) 1307 as it is received, and/or stored in the storage device 1310, or other associated non-volatile media, for later execution. In this manner, a processing unit may receive messages, data and/or program code in the form of a carrier wave.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. For example, the reader is to understand that the specific ordering and combination of process actions shown in the process flow diagrams described herein is merely illustrative, and the invention can be performed using different or additional process actions, or a different combination or ordering of process actions. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.